Proposal for Parallelizing the Unix Proc Structure
Revision: 1.3

Tera Computer Company
2815 Eastlake Avenue East
Seattle, WA 98102

June 29, 1993

1 Introduction

This is a proposal for parallelizing the proc structures in the Unix kernel. The goals were to have a method that required a minimum of recoding of Unix code and could scale well.

2 Current Implementation

The current BSD implementation of proc structures has a hash table for lookups and uses `pfind()` and `curproc` for acquiring proc entries. Since BSD4.4 is not a parallel Unix, the code assumes the kernel is a monitor and only one thread of control will ever be modifying the proc structure.

Each proc structure can reside on numerous lists: the hash table, the `allproc` list, the sibling list, the process group list, run/sleep queue, zombie list, etc.. The proc is created in `fork()` and freed in `wait()`.

3 Tera Proposal

3.1 Data Structures

Tera adds to each proc structure a reader/writer lock `p_rwLock` and a reference count `p_refCount`. The current Unix hash table is replaced with a supervisor level hash table. The hash table provides a lock per hash bucket, which allows for synchronization of the proc reference count. The C++ implementation of `HashChainExt_k` can be used as a base for the hash implementation. The link that points to the proc itself should be doubly linked so that `pRelease` won't have to follow links if the entry needs to be removed.

3.2 Proc Creation

Procs are created during `fork()`. The `fork()` code will allocate a new proc structure, initalize it, then add it to the proc hash table. When the proc is created it will have a reference count of 1. When the reference count goes to 0, the proc can be freed.

3.3 Proc Access

Currently Unix uses `pfind(pid)` and `curproc` to obtain a proc pointer. We propose to replace `curproc` with a macro that expands to `pfind(thisChore() -> myTaskIndex())`. This call is not strictly needed since the process should not be freed while its making a system call on itself but it might be useful for debugging.

The `pfind` routine will increment the proc's reference count while a new routine, `pRelease()` will decrement it. The latter must be placed in the Unix code to correspond to each `pfind()`; this will require following a lot of code paths and could be a major source of work and possible bugs. When the reference count goes to 0, `pRelease()` removes the proc from the hash list but does not free the proc; that is done in `wait`. Once a process becomes a zombie, `pfind()` should not return it. When a chore 'sleeps', it must release its proc lock and do a `pRelease`. Subsequently, when it awakes, it must do a `pfind` and regain its lock.

3.4 Proc Destruction

Currently Unix frees procs in the `wait` system call. Tera will do this as well, but only when the reference count becomes 0.

The pseudo-code looks like:

```
/* Unix Code */
ProcHashChainExt procHash; /* global instance of hash table */

int fork() {
    ... other fork code ...
    taskFork(); /* get taskID */
    allocate proc
    initialize proc
    pAdd(p);
    ... other fork code ...
}

struct DLListLink {
    void * item;
    DLListLink* next;
    DLListLink* prev;
};

struct HashBucket {
    SpinLock_k cs$;
    DLListLink* head;
};
```

```
/* Add proc to hash table */
void pAdd(struct * p) {
    HashBucket* bucket = &procHash[hash(p->p_pid)];
    ProcListLink* link;

    link = MALLOC(ProcListLink);
    bucket->cs$.enter(); /* locks the bucket */
    link->next = bucket->head;
    link->prev = NULL;
    bucket->head = link;
    if (bucket->head)
      bucket->head->prev = link;
    p->p_refCount++;
    bucket->cs$.exit();   /* unlocks the bucket */
}

/* Find proc with given pid and return a pointer to it. Increment ref count */
struct proc* pfind(int pid) {
    struct proc *p = NULL;
    DLListLink* link;
    HashBucket* bucket = &procHash[hash(pid)];

    bucket->cs$.enter(); /* locks the bucket */
    link = bucket->head;
    while (link != NULL && (struct proc*)(link->item)->p_pid != pid)
      link = link->next;
    if (link != NULL) {
        p = (struct proc*)(link->item);
        if (p->p_stat == SZOMB) /* don't return zombies */
          p = NULL;
        else
          p->p_refCount++;
    }
    bucket->cs$.exit();   /* unlocks the bucket */

    return p;
}
```

```
/* Decrement reference count of proc. Remove it from hash table when */
/* ref cnt goes to 0 */
void pRelease(struct proc* p) {
    HashBucket* bucket = &procHash[hash(p->p_pid)];

    bucket->cs$.enter(); /* locks the bucket */
    DLListLink* link = bucket->head;
    while (link != NULL && (struct proc*)(link->item)->p_pid != pid)
      link = link->next;
    if (link != NULL) {
        p->p_refCount--;
        if (p->p_refCount == 0) {
            assert_k(p->p_flag == SFREE);
            ...remove link from doubly linked list...;
            ...Post Event...;
        }
    }
    bucket->cs$.exit();   /* unlocks the bucket */
}

int wait() {
    ... other wait code ...
    p->p_flag &= SFREE;
    pRelease(p);
    /* stream blocks until all references are done */
    ... Wait on Event;...
    FREE(p);
    ... other wait code ...
}
```

3.5 Locking the Proc

A reader/writer lock was selected because its clear that many parts of the Unix kernel need read only access. However, to correctly determine which code paths require locks will require a lot of work. To allow this to be done over a period of time and as needed, initially **all** locks will be writer locks; this effectively makes it a spin lock. As people work and become more familiar with the code, they can change from writer to reader locks where valid. The locks still have to be released in the correct places. One way to do this is to add it to `pRelease()`.

3.6 Traversing Proc Lists

Proc structures can be on many lists: parent, sibiling, etc.. Traversing these lists requires changing the reference counts as well as locking the proc. By replacing proc pointers with process ids to be used with `pfind()`, reference counts remain correct. The downside is that traversals incur the cost

of a pfind() lookup for each link. Since most of these lists are not that long and the traversals are not that frequent, the added expense is worth the ease in parallization.

Here are the list of proc pointers and their fates:

- `p_link`: doubly-linked run/sleep queue pointer. Goes away; replaced by using task control block.
- `p_rlink`: doubly-linked run/sleep queue pointer. Goes away; replaced by using task control block.
- `p_nxt`: linked list of active and zombie procs. Goes away, replaced by using task control block.
- `p_prev`: linked list of active and zombie procs. Goes away, replaced by using task control block.
- `p_hash`: hashed based on `p_pid` for kill+exit+. Goes away, replaced by using links in proc hash table.
- `p_pgrpnxt`: pointer to next process in process group. Replaced by a process id.
- `p_pptr`: pointer to process structure of parent. Replaced by a process id.
- `p_osptr`: pointer to older sibling processes. Replaced by a process id.
- `p_ysptr`: pointer to younger siblings. Replaced by a process id.
- `p_cptr`: pointer to youngest living child. Replaced by a process id.

4 Session and Process Groups

Although not part of the proc structure, these structures also need parallelization. A similar scheme for locking and using reference counts can be used for these data structures. The process group and session replace their pointers to a process with a process id. Each structure will have a read/write lock and a reference count.

4.1 Generating the PID

The Tera kernel generates task ids that double as process ids. This works well for process ids, but Unix also uses process id for process group ids. This poses the following problem. Suppose that processes A, B and C belong to process group A, and then process A exits. The process group id must remain unique in the process id name space until the process group is deleted. Thus, the Tera kernel must not reuse the id 'A' until the process group A goes away.

One proposal is not to remove the `TaskControlBlock` until both the process and process group (if one exists) with that id go away. This would prevent the kernel from reusing the id. The following

pseudo code gives a sample implementation.

```
int wait() {
    ... other wait code ...
    p->p_flag &= SFREE;
    int id = p->p_pid;
    pRelease(p);
    /* stream blocks until all references are done */
    ... Wait on Event;...
    FREE(p);
    if (markProcessGroupFree(id) == FALSE)
      taskComplete(id);
    ... other wait code ...
}
```

```
/* Remove proc from process group. Remove and free process group when empty */
leavepg(struct proc* p) {
    hash(p->p_pid);
    lock bucket;
    find p's process group;
    remove process from process group;
    if process group is now empty {
        if (pg is marked to be freed)
          taskComplete (process group id);
        Free(process group);
    }
    unlock bucket;
}
```

```
/* Mark process group to call taskComplete(pid) when freed */
void markProcessGroupFree(int id) {
    bool ret;
    hash(id);
    lock bucket;
    search for process group with id;
    if process group is found {
        mark process group;
        ret = TRUE;
    } else
      ret = FALSE;
    unlock bucket;
    return ret;
}
```